INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Solving losely coupled Constraints

Michel Le Borgne

N° 6958

Juin 2009

Thème BIO

ISSN 0249-6399 ISRN INRIA/RR--6958--FR+ENG

Solving losely coupled Constraints

Michel Le Borgne

Thème BIO — Systèmes biologiques
Équipe-Projet Symbiose

Rapport de recherche n° 6958 — Juin 2009 — 19 pages

Abstract: Many practical problems are modelled by mean of constraints which, pairwise, share only a few variables if any. Such constraints are loosely coupled. In this paper we show how to use this property to lower the complexity of decision procedures or solver. The introduction of a data structure, the well formed covering tree allows for a reparametrisation of constraint problems in a way which can be seen as a generalisation of the triangulation procedure for system of linear equations.

Key-words: constraints, quantifier elimination

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Résolution de contraintes faiblement couplées

Résumé : Dans beaucoup de problèmes pratiques, on est amené à introduire des modèles basés sur de nombreuses relations définissant un ensemble de contraintes. De par la nature même du problème ou parce que le modélisateur est un humain, les contraintes, prises deux à deux, ne partagent qu'une faible partie voire aucune de leurs variables. La famille d'algorithmes présentée ici utilise cette propriété pour abaisser la complexité des problèmes de décision ou de résolution de tels systèmes de contraintes. Avec l'introduction des arbres couvrants bien formés, on arrive même à une reparametrisation des problèmes initiaux d'une manière qui peut être considérée comme un généralisation de la triangularisation des systèmes d'équations linéaires.

Mots-clés : contraintes, élimination des quantificateurs

1 Motivations

1.1 Context

In numerous domains, formal models are based on a set of constraints and predictions of the models are based on constraint satisfaction. Let us mentioned various applications of bounded model checking either in circuit conception or automatic control. Another domain where constraints are the model building bricks is qualitative modelling either through influence graphs or more dynamical models. Practical applications range from qualitative physic to system biology.

A constraint may be seen as a relation $R(x_1,\ldots,x_n)$ among some variables. Variables take values in various domains, which are in general finite. Due to the very nature of the modelled objects or to the fact that there are human creations, it appears that in many models based on set of constraints, each constraint involves only a small number of variables. As a consequence, two constraints share a very limited number of variables. For that reason, we will call such systems as *loosely coupled constraints systems*.

This characteristic of such constraint systems comes very naturally from the modelling process. We do not suggest here any preleminary transform of the set of constraints. However the most used transform is to replace a finite set of constraints by their conjunction. This transformation give rise to a new constraint which involves much more variables. In general, the representation of the new constraint is much more complex than the representation of the set of constraints of which it is the conjunction. Despite this drawback, replacing a set of constraint by their conjunction was current practice in algorithms based on BDD for boolean constraints. The reason was that, before the so called 'Sat revolution', no other practical procedure was known for deciding if the set of constraints can be satisfied and to produce a solution.

On the opposite, the use of SAT solvers necessitates a preliminary transform of the set of constraints into a set of clauses. Since each raw constraint must be decomposed in a conjunction of clauses, the new set of constraints is no more loosely coupled although it can be partitioned into subsets such that two constraints taken from two different sets are loosely coupled.

1.2 Example: boolean constraints

A boolean function $f(X_1,\ldots,X_n)$ is a function from $\{t,f\}^n$ into $\{t,f\}$. A boolean vector $(x_1,\ldots,x_n)$ satisfies a set of boolean functions $\{f_1,\ldots,f_p\}$ (seen as constraints) if for each k, $f_k(x_1,\ldots,f_n)=t$. The set of constraints $\{f_1,\ldots,f_p\}$ is satisfiable if there exist a boolean vector which satisfies it. Remark that satisfying a set of constraints $\{f_1,\ldots,f_p\}$ is equivalent to satisfying the unique constraint $f_1\wedge f_2\wedge\ldots\wedge f_n$.

Going further, an equivalent point of view is to say that the formula

$$\exists X_1\exists X_2\ldots\exists X_n f_1\wedge f_2\wedge\ldots\wedge f_n(X_1,\ldots,X_n)$$

without free variables is true. It happens that quantifier elimination is possible in boolean formula and very easy. Quantifier elimination consist in replacing the formula $\exists X_1 f(X_1,\ldots,X_n)$ by a formula without quantifier and without the variable X_1.

The equivalent formula is $f(X_1 = t, X_2, \ldots X_n) \vee f(X_1 = f, X_2, \ldots X_n)$. So the operation is easy for a single simple formula and a few variables. When one deals with numerous constraints and many variables, it becomes rapidly intractable. The main obstacle is the computation of the conjunction of formulas which is intractable most of the case.

The number of variables is a predominant factor in the complexity of computations on boolean formulas. Memory requirement and computation time increase rapidly with the number of variables, even with an efficient representation of boolean formulas as is the BDD (Boolean decision diagram) one. So the main idea of our algorithm is not to perform conjunction of formulas then quantifier elimination but to interleave the two operations in order to compute conjunction on formulas with few variables. The key observation is: if X is not a variable of f then $\exists X\, f \wedge g$ is equivalent to $f \wedge \exists X g$. If we perform quantifier elimination in $\exists X g$, the second conjunction deals with simpler formulas.

A more illustrative example is given by three loosely coupled formulas:

$$\exists X_1 \exists X_2 \exists X_3 \exists X_4 \;\; f_1(X_1, X_2) \wedge f_2(X_2, X_3) \wedge f_3(X_3, X_4)$$

is equivalent to

$$\exists X_1 (\exists X_2 (f_1(X_1, X_2) \wedge (\exists X_2 f_2(X_2, X_3) \wedge (\exists X_4 f_3(X_3, X_4)))))$$

If the last formula computation is performed as indicated by parenthesis, it computes conjunctions and quantifier elimination with formulas with at most two variables instead of four if the first formula were computed as it.

Although it is not evident from examples, another key property of quantifier elimination which is used in our algorithm is commutativity. This property gives a large degree of freedom in the choice of the variable to be eliminated at each step. If we think of an extension of this algorithm to more general quantifier elimination problems such that those encountered in QDF(Quantified Boolean Formula) decision problems, we will face difficulties due to the non commutativity between existential and universal quantifiers.

1.3 Generalization

Existential quantifier elimination may be seen as a projection operation. In the case of boolean constraints, it gives the constraints that must be satisfied by the remaining variables in order to find a solution for the original constraints. Looking at the computation of the formula without quantifier, it appears also as a transform similar to the computation of a marginal distribution. In the case of boolean constraints, the **or** operation play the same role than the sum in the computation of a marginal of a discrete probability distribution. The **and** operator plays a role similar to the product in probability distribution. This shows that we are in the so called sum-product paradigm which extends to many situations.

It is well known that the algorithms derived in the sum-product paradigm apply also to optimization problems provide that the cost function has some nice property. In system biology, cost minimization is used for inferring biological networks of influence. So far, algorithms used for that goal are iterative algorithms which are known

to converge to a an approximate solution. Moreover, a set of experimental data can be explained by several influence graphs. An important information is whether an influence inferred is necessary or may be replaced by an alternative one. We will see that the proposed algorithm compute exact solutions and detect necessary influences.

For that many reasons, we will present our algorithm in a very general setting. The next section introduces the general satisfiability problem together with notations used throughout the paper. The two preceding examples, boolean constraints and cost optimization will be used as illustrations of the main concepts. The following section is devoted to dependence graphs and well covering trees, two key concepts at the origin of the algorithms. In next section we will prove the general algorithm and give some interesting practical consequences of the main theorem. Finally, we show that our algorithm gives a new representation of a set of constraints.

A prototype version of the algorithm is implemented in the BIOQUALI package. BIOQUALI is a Python module that gives a framework for modelling biological systems as influence graphs. The introduction of this algorithms improved BIOQUALI efficiency by many orders of magnitude.

2 The generalized satisfiability problem

2.1 Generalized constraints

Let $(D_i)_{i=0,\ldots,n}$ and D a family of domains. Each domain is in general finite but all the domains are not necessarily identical. Associated with the family (D_i) we consider variables $X_0, X_1, \ldots, X_n$ with corresponding assignments $x_0, x_1, \ldots, x_n$ in the domains $D_0, \ldots, D_n$ respectively.

A constraint is a pair (f, P) where $f(X_0, X_1, \ldots, X_n)$ is a function from $\Pi_{i=0}^{i=n} D_i$ into D and $P(z)$ a logical formula with one free variable z which can be interpreted on D. In the following, this formula will be designated as the predicate of the constraint problem.

Definition 2.1. A multiple $(x_0, x_1, \ldots, x_n)$ satisfies (f, P) if $P(f(x_0, x_1, \ldots, x_n))$ is true. The satisfiability set of (f, P) denoted $sat(f, P)$ is the subset of multiples $(x_0, x_1, \ldots, x_n)$ in $\Pi_{i=0}^{i=n}$ satisfying (f, P).

When there is no ambiguity, we will drop the predicate P.

Example 3. In the case of boolean constraints, the domains D_i and D are the set of booleans $B = \{true, false\}$. The function f is a boolean function and $P(Z) = \{Z = true\}$. The notion of satisfiability is the usual one. Depending on the application we are either interested in checking the satisfiability property or in computing all the multiples satisfying the constraints. When f is a constant function, it is satisfiable if and only if it is equal to $True$

Example 4. If we consider an optimization problem, for example cost minimization on the product of boolean spaces $\Pi_{i=0}^{i=n} B$, the function f is a cost function in the domain of real numbers R for example. Let us denote it as C instead of f to distinguish between our two examples. With $P_c(Z) = \forall y\ C(y) \geq Z$, a multiple $(x_0, x_1, \ldots, x_n)$ satisfies

(C,P_c) if it is a point where the cost function is at his minimum. The satisfiability set is then the set $argmin(C)$. Since the domain of the cost function is finite, the satisfiability problem has always a solution. In this case we are more interested in the computation of the satisfiability set or at least of an element of this set as an example. Notice also that a constant function is always satisfiable for an optimization problem, whatever the value.

Definition 4.1. Two constraints (f_1,P_1) and (f_2,P_2) defined on the same domain are equivalent if they have the same satisfiability set.

This definition is a straightforward generalization of the definition of equivalent formulas in propositional logic. Considering two different predicates is often interesting. For example, in some applications we transform an optimisation constraints into a logical constraints by reparametrization of the satisfiability set.

Some components of the domain $\prod_{i\in I} D_i$ have interesting properties with respect to satisfiability sets.

Definition 4.2. Given a constraint $(f(X_1,\ldots,X_i,\ldots X_n),P)$, a component X_i is a hard component if for all multiple of values $(x_1,\ldots,x_i,\ldots,x_n)$ satisfying (f,P), x_i takes the same value.

The concept of hard component comes from the field of qualitative modelling. For a qualitative model represented by constraints, a hard component represents a prediction of the model. If a minimization or maximization problem originates in a statistical inference technique, a hard component represents a reliable inference of parameter.

4.1 Variable elimination

The second ingredient we found in boolean constraint is an operation named quantifier elimination. We generalized it as variable elimination. With the previous notations if f is a function and X_i is a variable, we assume that we have an operation called variable elimination transforming the function f into a function denoted $\xi X_i f(X_0,\ldots,X_{i-1},X_{i+1},\ldots,X_n)$ defined on $D_0\times\ldots\times D_{i-1}\times D_{i+1}\times\ldots\times D_n$. Associated with the same predicate P, the new function defines a new constraint on a smaller space. Of cause, we assume that the operation can be repeated by choosing a variable to eliminate among the remaining ones. The minimum property of this variable elimination operation is some kind of commutativity.

Property 4.3. *For all choices of different variables X_i and X_j, the two constraints $\xi X_i\xi X_j f$ and $\xi X_j\xi X_i f$ are equivalent.*

Example 5. For the boolean constraints, variable elimination is existential quantifier elimination defined by the simple formula: $\xi X_i f = f_{|X_i=true} \vee f_{|X_i=false}$. It is well known to be commutative.

Example 6. Without more information it is not evident to guess what could be a useful variable elimination for a minimization problem modelled as a generalized constraint problem (C,P_c). If one recalls that variable elimination has some similarity with marginal distribution computation, the following operation looks satisfactory:

$$(\xi X_i C)(x_0,\ldots,x_{i-1},x_{i+1}\ldots x_n) = min_{x_i\in D_i} C(x_0,\ldots x_n)$$

The commutativity is trivially satisfied.

Commutativity is not a sufficient property. In the case of quantifier elimination for boolean constraints, it is truly a projection. That means that it has something to do with satisfiability. A multiple y satisfying a projected constraint $\xi X f$ can be lifted into a multiple (x,y) satisfying the original constraint f. We impose this property to variable elimination:

Property 6.1. *Variable elimination must be a projection that is to say if y satisfies $(\xi X f, P)$ if and only if there exist an x in its domain such that (x,y) satisfies (f,P).*

y denote a multiple which is an assignment of the variables of $\xi X f$.

Example 7. The variable elimination associated to existential quantifier elimination in boolean constraints is a projection. It is the prototype of this concept.

Example 8. We show here that our choice for ξXC in the case of a cost minimization problem has the projection property. So let y satisfying $(\xi XC, P_c)$. That means:

$$\forall y' \ \xi XC(y') \ \geq \ \xi XC(y)$$

From the definition of ξXC we get:

$$min_{x\in D_x} C(x,y') \ \geq \ min_{x\in D_x} C(x,y) \ = \ C(x_0,y)$$

where D_x denotes the finite domain of x and x_0 a point where the *min* is reached. Now let $x' \in D_x$. Then:

$$C(x',y') \ \geq \ min_{x\in D_x} C(x,y') \ \geq \ C(x_0,y)$$

which shows that (x_0,y) satisfies (C,P_c).

8.1 Decomposition of functions

Now we turn to function decomposition. In our motivating example, we emphasized on the fact that a decomposition of a constraint in a conjunction of loosely coupled constraints might improve satisfiability checking. We need now to generalize this idea of decomposition. For that purpose we introduce an operation $\odot$ on the domain D. From the motivating example it appears that commutativity and associativity are very useful.

Property 8.1. *The operator $\odot$ on D is commutative and associative.*

The operation on D extends classically to functions from $\Pi_{i=0}^{i=n} D_i$ into D. From now we will assume that f decompose into several functions: $f \ = \ \odot_{k=0,..,p} f_k$. Each function f_k depends only on a subset of variables of f. We impose also some kind of compatibility between variable elimination and the $\odot$ operation:

Property 8.2. *If X is not a variable of f then $\xi X(f(Y) \ \odot \ g(X,Y))$ is equivalent to $f(Y) \ \odot \ \xi X g(X,Y)$*

Example 9. For boolean constraints the $\odot$ operator is the conjunction $\wedge$. It has the desired properties.

Example 10. Additive cost function are very often encountered. So we will consider the sum on real numbers as $\odot$ operator in this example. It is clearly commutative and associative. Moreover, it is not difficult to check that $\xi X(f(y)+g(x,y)) = min_x(f(y)+g(x,y) = f(y)+min_x g(x,y) = f(y)+\xi X g(x,y)$

Since predicates used in the definition of satisfiability are predicates on constants, we need a property to compute satisfiability of expressions on constants.

Property 10.1. *Given a predicate P and an operator $\odot$ defined on the same domain, then P and $\odot$ are compatible if $P(x \odot y)$ is satisfied if and only if $P(x)$ and $P(y)$ are satisfied.*

Example 11. This condition is satisfied for boolean constraint since $\odot$ is the conjunction.

Example 12. For optimisation problems, $P(x)$ is satisfied for any constant x and the property is trivialy satisfied since the two members of the equivalence are always $True$.

This property together with property (8.2) gives a key proposition for reparametrisation of a set of constraints:

Proposition 12.1. *With previous notations consider a constraint $f = f_1 \odot f_2 \odot \ldots \odot f_n$ such that for all i, j with $i \neq j$, f_i and f_j have no common variable. Then f is satisfied if and only if each f_i is satisfied.*

Lastly, for initializing recursive algorithms, we need an identity element. So, in the following we assume:

Property 12.2. *$\odot$ has an identity element denoted ε and $P(\varepsilon)$ is satisfied.*

12.1 Satisfiability of a set of constraints

Given a set of propositional formulas, the satisfiability of the set is usually defined as the possibility to find an assignment of propositional variables that satisfies all formulas. This way of defining satisfiability of a set of constraints doesn't generalize to minimization problems for example. For that reason we prefer the equivalent definition which says that a set of propositional formulae is satisfiable if the conjunction of the formulas is satisfiable. This definition generalizes easily by replacing the conjunction operation on propositional formulas by the generic composition $\odot$ of functions:

Definition 12.3. With the preceding notations, a finite set of constraints $\{(f_i,P)_{i\in I}\}$ is satisfiable if the constraint $(\bigodot_{i\in I} f_i, P)$ is satisfiable.

In the minimization problem, this definition says that the minimization of the set of cost functions $(C_i(X))_{i\in I}$ is obtained at x when the cost function $\sum_{i\in I} C_i(X)$ has a minimum at x. Of course, that doesn't mean that every partial cost $C_i(X)$ is minimized.

The following proposition link satisfiability and variable elimination.

Proposition 12.4. *A set of generalized constraints $\{(f_i,P)_{i\in I}\}$ is satisfiable if and only if the unique constraint with one variable $(\xi X_1 \ldots \xi X_{i-1} \xi X_{i+1} \ldots \xi X_n \bigodot_{i\in I} f_i, P)$ is satisfiable for some variable X_i.*

The proof is immediate by repeated application of property 6.1.

A more logical formulation would be to eliminate all variables. However we are most interested in finding a solution and the formulation of proposition 12.4 is more appropriate. Remark that in minimization problems, there is always a solution, hence the satisfiability is granted. The hard part is to find a point where the optimum of a cost function is obtained. In the following we will describe an optimization algorithm, which provides examples, starting with one variable functions.

The computation of all but one variable elimination is also a method to detect hard components of a set of constraints. It is enough to show that the one variable constraint has only one solution. We will also see later that it gives an interesting reparametrization of the set of constraints.

13 Dependence graphs and well formed covering trees

As shared variables between constraints are important for variable elimination, we introduce a graph representation of these links.

13.1 Dependence graph

A dependence graph G is a bipartite graph whose nodes are the variables x_i on one hand and the constraint components f_j on the other hand. There is an edge between the variable x and the function f if x is a variable of f. $V(G)$ represents the vertices of G which are variables and $F(G)$ the vertices which are functions. A dependence graph is

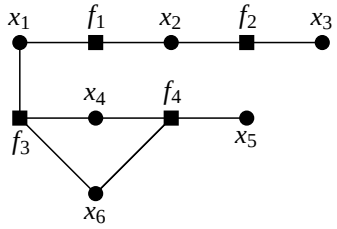


Figure 1: A dependance graph

not necesseraly connected. However, satisfiability of a conjunction of functions must be checked on each component. More precisely:

Proposition 13.1. *Let $\{f_i(X)\}_{(i\in I)}$ a finite family of functions defining a composite constraint $C(X) = \bigodot f_i(X)$. If $G = \cup_{j\in J} G_j$ is the partition of the associated dependence graph into connected componants, let $C_j(X) = \bigodot_{f\in F(G_j)} f(X)$. Then $C(X)$ is satisfiable if and only if each $C_j(X)$ is satisfiable.*

The set of functions $\{f_i(X)\}_{(i\in I)} = F(G)$ can be partitioned as $F(G) = \cup_j F(G_j)$, the constraint $C(X)$ can be decomposed as $C(X) = \bigodot_j C_j(X)$ with the local constraints $C_j(X) = \bigodot_{f\in F(G_j)} f(X)$. Now the variables of the functions in $C_j(X)$ are connected to the functions in C_j and consequently belong to the same connected component. This

implies that C_j and C_k have no common variables if $j \neq k$. The proposition is an immediate consequence of this fact and proposition 12.1.

13.2 Well formed covering trees

Covering trees of undirected graphs are generally undirected trees. However, since we are looking for an elimination order of variables, we will consider *oriented* covering trees. Since dependence graphs are bipartite, their covering trees are also bipartite. With oriented trees we can use the standard notions of *descendant* and *ancestor* as the notions of *child* and *father*.

Definition 13.2. A well formed covering tree of a dependence graph G is a directed covering tree such that:

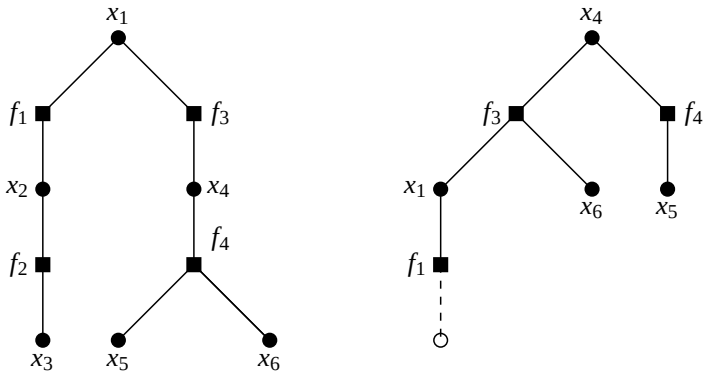
1. the root is a variable vertex of G.
2. for each function node f, a variable of f is:
   - either a child of f
   - either a variable ancestor of f
   - or a child of a function node which is an ancestor of f

For each variable, there is at least one well formed covering tree with root the chosen variable.

Any traversal of a graph discover a directed covering tree. During a traversal some vertices are discovered and wait to be used to explore further the graph. Let A be the set of discovered not yet used vertices in a traversal of a dependence graph. It is well known that deep first traversal is obtained if A is managed as a stack and breath first traversal if A is managed as a FIFO. Unfortunately, neither deep first traversal, nor breath first traversal give a well formed covering tree as illustrated in figures 2(a) and 2(b) which shows deep first and width first examples of traversal of the preceding dependance graph. In figure 2(a), x_6 which is a variable of f_3 is not in correct position with respect to f_3. In figure 2(b), x_6 which is also a variable of f_4 is not in correct position with respect to f_4.

Well formed covering trees are obtained by a mixture of breath first and deep first traversal. More precisely, A is managed as a stack. The variable x chosen to be the root is pushed on the stack. During the traversal, function nodes are pushed individually. When such a function node f is discovered and push on the stack, all the variables of f which are not already discovered are pushed on the stack. So the proposed traversal behaves as a deep first one on variable nodes and as a breath first one on function nodes. Moreover, variables nodes and function nodes are added to the tree when they are discovered. Since the graph is a bipartite one, variable nodes are always discovered as successors of function nodes.

Proposition 13.3. *Mixed traversals, as described above, build well formed covering trees of a dependence graph.*



(a) Deep first traversal (b) Width first traversal

Figure 2: Standard traversals

Let f a function node in the tree resulting from a mixed traversal. The father node of f is a variable of f and is clearly an ancestor. Children of f are also variables of f. So consider a f variable x which is neither a child nor the father of f.

Since x is not a children of f, it was already in the tree when f was added. If x were not in the stack when f was put in the stack, then x is necessary a neighbor of f which was used to discover f. This implies that x is the father of f in the tree, against our hypothesis.

So x was yet in the stack when f was discovered. Let g the father of x. g is the node from which x was discovered and put in A. g cannot be a descendant of f because all descendants of f are descendants of children of f. Since all variables of f are in the tree after the children of f are put in the tree, x cannot be discovered later.

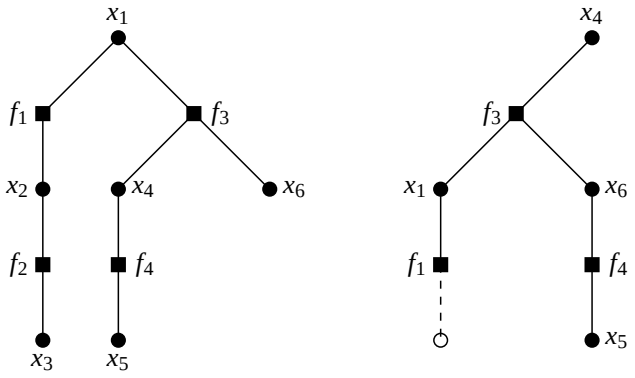
If g is not an ancestor of f, consider the variable node y which is the root of the smallest subtree containing both f and g. Such a variable node exist since the root of the whole tree is a variable. If the branch containing f were built before the branch containing g, x was discovered before g as a children of f. If the branch containing g were built before those containing f, f must be a successor of x i.e. x the father of f. Both hypothesis leads to a conclusion contradicting our assumption on x. So g is necessary an ancestor of f.□

Figure 3 shows two well formed covering trees of the graph of figure 1.

In order to ease the algorithm description and the following proofs, we introduce some definitions.

Let n be a node in a well covering tree. We denote by $DV(n)$ all the variable nodes which are descendant of n in the covering tree, n being excluded. Similarly, $DF(n)$ will denote all the function nodes which have n as ancestor with n excluded.

Similarly we denote by $AF(n)$ all the function nodes which are on the unique path from n to the root with n excluded again. For variables we need a different definition adapted to the particular structure of well covering trees. We denote by $AV(n)$ the variables which are either on the path from n to the root or are children of a function node which is on this path. The variables in $AV(n)$ will be said as being above n.



(a) Root x_1 (b) Root x_4

Figure 3: Well covering trees

The following lemma is very useful:

Lemma 13.4. *Let x a variable and f,g two functions with a common variable y which is in the subtree with x as root. Then f and g are in the same child of x.*

If *y* is a child of f then *g* is in the subtree with root f since it cannot be a child of *g* and consequently *y* is above *g*. The similar case when *y* is a child of *g* leads to the same conclusion and in both cases the lemma is verified.

Assume now that *y* is above f. Then *y* and f are in the same child of *x*. If *y* is a child of *g*, *y* and *g* are in the same child of *x*. If *y* is on the path from *g* to the root of the well covering tree, then this path contains *x* since *y* is in the subtree with root *x* so *g* and *y* and f are in the same child of *x*.

The last case is when *y* is the child of *h* which is on the path from *g* to the root. But in this case, *h*, is on the path from *y* to the root which contains *x* and since *y* is a child of *h*, *x* is on the path from *h* to the root. So *y* and *g* and of cause f are on the same child of *x*. □

This lemma is more useful on the following form: if f and *g* are on different children of *x* then their common variables are in $\{x\} \cup AV(x)$.

Lemma 13.5. *Let X representing the variables of a well covering tree and $CH(f)$ the children of a function node. Then if $(f_i)_{i\in I}$ is the family of function nodes of the tree, then $(\{x_0\}, (CH(f_i))_{f\in I})$ is a partition of X.*

Since the tree is a covering tree of the dependence graph, all the variables appear in the tree. The sets $CH(f_i)$ are pairwise disjoints because two function nodes cannot have a common child.□

14 Checking consistency

Checking the consistency of a set of constraints $\{(f_i, P)_{i\in I}$ can be done by eliminating all but one variables. As we suggested in section 1.2, a cleaver choice of the order of

variable elimination can dramatically reduce the complexity of this elimination. This section is devoted to the description and the proof of such an algorithm.

In this section we assume that the dependence graph of our set of constraint is connected. In view of proposition 13.1, if it were not the case, satisfiability of the set of constraints must be checked on every connected component.

14.1 The algorithm

The algorithm is a message passing algorithm living on a well formed covering tree. In order to describe it, we have to introduce some notations. If U is a set of variables, $\xi(U)$ denotes the successive elimination of all the variables in U. This notation is not ambiguous since eliminations commute. Given a well formed covering tree, for each node n we define a constraint $margin(n)$ recursively:

$$\text{if } x \text{ is a variable node: } margin(x) = \bigodot_{f \in CH(x)} margin(f)$$

$$\text{if } f \text{ is a function node } margin(f) = \xi(CH(f))(f \odot \bigodot_{x \in CH(f)} margin(x))$$

To be complete we have to specify stopping cases. These cases happen when a node has no child. If it is a variable node x then $margin(x) = \varepsilon$ the identity element of the $\odot$ operator. If we have a function node f then $margin(f) = f$.

Theorem 14.1. *For all variable node x of a well covering tree:*

$$margin(x) = \xi DV(x) \bigodot_{f \in DF(x)} f(X)$$

and for all function nodes f:

$$margin(f) = \xi DV(f)\,(f \odot \bigodot_{g \in DF(f)} g(X))$$

The proof is by induction on the height of the nodes in the tree. If x is a variable leaf, $margin(x) = \varepsilon$ and the proposition is trivially true since the operator $\odot$ applied to an empty set of operands gives ε, the neutral element. If f is a function node, $margin(f) = f$ and the proposition is satisfied for the same reason.

Assume that the proposition is satisfied for all nodes with a height lower than m. Let n be a node with height m. If n is a variable node x, then $margin(x) = \bigodot_{f \in CH(x)} margin(x)$. The children of x are at a height lower than m and the hypothesis applies:

$$margin(x) = \bigodot_{f \in CH(x)} \xi DV(f)(\ f \odot \bigodot_{g \in DF(f)} g(X))$$

Now let g_1 and g_2 be two function nodes in $DF(x) = \bigcup_{f \in CH(x)} DF(f)$ respectively in $DF(f_1)$ and $DF(f_2)$ where f_1 and f_2 are two different children of x. If y is a variable

common to g_1 and g_2, it must be a parent of x or a child of a fonction node which is a parent of x since the tree from which it's a node is a well formed covering tree. So if f_1 and f_2 are children of x, $DV(f_1) \cup DF(f_2) = \emptyset$ and:

$$\begin{aligned} margin(x) &= \xi(\bigcup_{f\in CH(x)} DV(f)) \bigodot_{f\in CH(x)} f \odot \bigodot_{g\in DF(f)} g(X) && (1)\\ &= \xi DV(x) \bigodot_{f\in CH(x)} f \odot \bigodot_{g\in DF(f)} g(X) && (2)\\ &= \xi DV(x) \bigodot_{f\in DF(x)} f(X) && (3) \end{aligned}$$

since $DF(x) = \cup_{f\in CH(x)} DF(f)$.
Now if n is a function node:

$$margin(f) = \xi CH(f)(f \odot \bigodot_{x\in CH(f)} margin(x))$$

and the recurrence hypothesis applies to $margin(x)$ giving:

$$margin(f) = \xi CH(f)(f \odot \bigodot_{x\in CH(f)} \xi DV(x) \bigodot_{g\in DF(x)} g(X))$$

From the properties of well formed covering trees, the sets $DV(x)$ doesn't contain x and are disjoint. So it is possible to move the different elimination operations outside the composition:

$$\begin{aligned} margin(f) &= \xi CH(f)(f \odot \xi(\cup_{x\in CH(f)} DV(x)) \bigodot_{x\in CH(f)} \bigodot_{g\in DF(x)} g(X)) && (4)\\ &= \xi CH(f)(f \odot \xi(\cup_{x\in CH(f)} DV(x)) \bigodot_{g\in DV(f)} g(X) && (5) \end{aligned}$$

Again, from properties of well formed trees, it is easy to show that no $DV(x)$ contains any variable of f if x is a child of f. So it is possible to move the elimination a step further:

$$\begin{aligned} margin(f) &= \xi CH(f) \xi(\cup_{x\in CH(f)} DV(x))(f \odot \bigodot_{g\in DV(f)} g(X)) && (6)\\ &= \xi DV(f)(f \odot \bigodot_{g\in DV(f)} g(X)) && (7) \end{aligned}$$

Corollary 14.2. *Given a well formed covering tree of the dependence graph of a set of constraints $(f_i(X))_{i\in I}$ with x the variable root, then margin(x) is a function with only one variable x and:*

$$margin(x) = \xi \hat{X} \bigodot_{i\in I} f_i(X)$$

where $\hat{X}$ is equal to $X - \{x\}$

This corollary gives a solution to the consistency problem if one knows how to solve it for one variable. This corollary gives also a mean to compute hard components of a set of constraints if one knows how to solve constraints with one variable. It is enough to build an x rooted well formed covering tree of the dependence graph and apply the algorithm. Solving the constraint given by $margin(x)$ detects if there is more than one solution. It is possible to perform this computation for all variables and select the hard components. Remark that in doing so, we do many redundant computations which should be possible to avoid, improving further the algorithm.

14.2 A recursive algorithm

To conclude this section we give a recursive algorithm that implicitly build a well formed covering tree and compute the *margin* function on a dependence graph of a set of constraints $(f_i(X))_{i\in I}$.

```
margin(n):
mark(n)
if n is a variable node then
   m := ε
   for all nv neighbour of n do
      if nv not marked then
         m := m ⊙ margin(nv)
      end if
   end for
   return m
else
   m :=n.func
   lvar := []
   l_neighbours := []
   for all nv neighbor of n do
      if nv not marked then
         l_neighbours.append(nv)
         mark(nv)
      end if
   end for
   for all nv in l_neighbours do
      m := m ⊙ margin(nv)
      lvar.append(nv.var)
   end for
   return ξ(lvar) m
end if
```

As any constraint and any variable apppears only once in a covering tree, the size of the tree is $O(m+n)$ where m is the number of constraints and n the number of variables. The complexity of the algorithm is essentialy dependant on the maximum complexity of the local computations: $margin(x) = \bigodot_{f\in CH(x)} margin(f)$ and

$margin(f) = \xi(CH(f))(f \odot \bigodot_{x \in CH(f)} margin(x))$ which are potentialy exponential in the number of variables involved. Cycles in the dependance graph increase this number of local variables and the size of cliques plays also a predominant role in complexity.

Properties of the dependance graph cannot be improved by the algorithms. However, as described above, this raw algorithm gives opportunities for heuristic optimisations. When exploring the unmarked neighbours of a function node or a variable node, we don't give any constraint on the exploration order. A carefull choice of the next node might improve the complexity of the algorithm. Various heuristics are presently under study.

15 Constraint reparametrization

In the preceding section, we have described an algorithm for checking consistency of a set of constraints. However, in many applications, consistency is not a sufficient information and at least an example of an n-tuple satisfying the constraints is needed. So we have to turn our algorithm into a solver. In the process, we will get an interesting reparametrization of the set of constraints.

Let us associate to each function node of a well covering tree, the constraint:

$$C_f = f \odot \bigodot_{x \in CH(f)} m(x) = f \odot \bigodot_{x \in CH(f)} \bigodot_{g \in CH(x)} m(g) \tag{8}$$

The constraint associated with the root x_0 is:

$$C_0 = \bigodot_{f \in CH(x_0)} m(f)$$

These new constraints can be partially ordered on a tree where C_g is child of C_g if and only if there exist a variable x which is a child of f and g is a child of x in the covering tree. C_0 is the root of the new tree with children $\{C_f / f \in CH(x_0)\}$. We will now show that this new tree gives a resolution order for the set of constraints $\{C_f\}$. For that reason we call the new tree, the *resolution tree R* associated with the well covering tree. Given a variable x we will denote by $\bar{x}$ an instantiation of this variable. We assume we know how to solve constraints when the number of variables is low.

From the construction of a well covering tree, the constraint C_0 has only one variable x_0. Let $\bar{x_0}$ be a solution. Now substitute $\bar{x_0}$ for x_0 in each child of C_0 in the resolution tree, which is also a child in the well covering tree. From well covering tree properties, it is not difficult to show that for two children f, g of x_0, $var(f) \cap var(g) = \{x_0\}$. After instantiation of x_0, the constraints $f|_{x_0=\bar{x_0}}$ and $g|_{x_0=\bar{x_0}}$ have no common variable. So they can be solved separately.

Since x_0 satisfies $C_0(x_0) = \bigodot_{f \in CH(x_0)} m(f) = \bigodot_{f \in CH(x_0)} \xi(CH(f)) C_f$ it satisfies the equivalent constraint $\xi(\bigcup_{f \in CH(x_0)} CH(f)) \bigodot_{f \in CH(x_0)} C_f$ obtained from repeated applications of property 8.2 which is possible, thanks to the separation of variables. Let $Y = \bigcup_{f \in CH(x_0)} CH(f) = \bigcup_{f \in CH(x_0)} Y_f$ where $Y_f = CH(f)$. From the projection property 6.1, there exist a $\bar{Y} = \bigcup_{f \in CH(x_0)} Y_f$ satisfying $\xi(\bigcup_{f \in CH(x_0)} CH(f)) \bigodot_{f \in CH(x_0)} C_f|_{x_0=\bar{x_0}}$. By separation of variables, Y_f satisfies $C_f|_{x_0=\bar{x_0}}$ and $(\bar{x_0}, Y_f)$ satisfies C_f.

The resolution process can be propagated along the resolution tree branches. Assume that $\bar{X}$ satisfies C_f for a node f of the resolution tree and X contains all the variables above f. In

$$C_f = f \odot \bigodot_{x \in CH(f)} m(x)$$

the variables in $CH(f)$ are variables of f and consequently from C_f. The fundamental property of well formed covering trees implies that in $C_f|_{X=\bar{X}}$, the component constraints $m(x)|_{X=\bar{X}}$ have separated variables. The propagation of resolution described for the root node in the preceding paragraph can be repeated from node x. Consequently, we have to solve separately $C_g|_{X=\bar{X}}$ for the variables $CH(g)$ and for all g in $CH(x)$. The projection property of variable elimination ensures that solutions exist.

In the process, we have to solve the constraints $C_g|_{X=\bar{X}}$ which involve the variables in $CH(g)$. When constraints are loosely coupled, the sets $CH(g)$ are small and finding a solution is not a complex task. For example, for boolean constraints, a BDD representation is well suited to quickly find a solution. So for the set of constraints $\{C_{f_i}\}$ derived from the set of constraint components f_i, it is relatively easy to find a solution. The following theorem turn our algorithm into a solver procedure:

Theorem 15.1. *Given a constraint $f = \bigodot_{i \in I} f_i$, and the derived constraints $\{C_{x_0}, C_{f_i}\}$, let denote by $A(f)$ the set of variables which are above f in a well formed covering tree associated with $f = \bigodot_{i \in I} f_i$. If $\bar{X}$ is a set of values for the variables X obtained by the resolution procedure described above, then $\bar{X}$ satisfies f.*

Conversely, if $\bar{X}$ satisfies f, it satisfies each constraint in $\{C_{x_0}, C_{f_i}|\bar{X}|_{A(f)}\}$.

Given a resolution tree R, a pruning of R is a tree T such that if n is a node of T then n is a node of R and the predecessor of n is in R. Let $L(T)$ the leaves of T and $DV(T)$ the variables of the leaves of T which are not variables of other nodes. Given an instantiation $\bar{X}$ of X, let us denote $\bar{X}|_T$ an instantiation of the variables in $\cup_{g \in T} AV(g)$. It is an instantiation of all the variables of the nodes of T except those in $DV(T)$. The interior of a tree T is defined as $\tilde{T} = T \setminus L(T)$. Finally, a full pruning of R is a pruning such that for all node n in the interior $\tilde{T}$ of T, the nodes $succ(n)$ are also in T.

Lemma 15.2. *If T is a full pruning of the resolution tree R then $\bar{X}|_T$ satisfies*

$$\bigodot_{h \in \tilde{T}} h \odot \xi DV(T)\Big(\bigodot_{g \in L(T)} C_g\Big) \tag{9}$$

The proof is by induction. The lemma is true when $T = \{x_0, CH(x_0)\}$ since in this case $\bar{X}|_T = \bar{x_0}$ and satisfies

$$C_0(x_0) = \bigodot_{g \in CH(x_0)} \xi CH(g) C_g = \xi DV(T) \bigodot_{g \in DV(T)} C_g \tag{10}$$

because the set of variables $CH(g)$ are pairwise disjoint and their union is $DV(T)$

Assume now T is a full pruning of the resolution tree R not equal to R and let $g_1 \in L(T)$ such that g_1 has at least one child in R. Then $\bar{X}|_T$ satisfies the formula:

$$\bigodot_{h\in\tilde{T}} h \odot \xi DV(T)(\bigodot_{g\in L(T)\setminus g_1} C_g \odot C_{g_1}) = \bigodot_{h\in\tilde{T}} h \odot \left[\xi DV(T\setminus g_1) \bigodot_{g\in L(T)\setminus g_1} C_g\right] \odot \xi CH(g_1) C_{g_1} \tag{11}$$

The last equality coming from the separation of variables in well covering trees.

Let $T' = T \cup succ(g_1)$. T' is a full pruning of R strictly containing T. The following properties are then easy to prove:

$$X|_{T'} = X|_T \cup CH(g_1) \tag{12}$$

$$L(T') = (L(T)\setminus g_1) \cup succ(g_1) \tag{13}$$

$$DV(T') = (DV(T)\setminus CH(g_1)) \cup \bigcup_{g\in succ(g_1)} DV(g) \tag{14}$$

Using the definition of C_g from 8, we can rewrite 11 as:

$$\bigodot_{h\in\tilde{T}} h \odot \left[\xi(DV(T)\setminus CH(g_1)) \bigodot_{g\in L(T)\setminus g_1} C_g\right] \odot \xi CH(g_1) \left[g_1 \odot \bigodot_{g\in succ(g_1)} \xi CH(g) C_g\right] \tag{15}$$

Since T is a pruning of a well formed covering tree, $X|_T$ contains the variables $AF(g_1)$ and consequently, from the resolution process, $\bar{X}|_{CH(g_1)}$ satisfies $C_{g_1}|_{\bar{X}_T}$. So, from 12, $\bar{X}|_{T'}$ satisfies:

$$\bigodot_{h\in\tilde{T}} h \odot g_1 \odot \left[\xi(DV(T)\setminus CH(g_1)) \bigodot_{g\in L(T)\setminus g_1} C_g\right] \odot \bigodot_{g\in succ(g_1)} \xi CH(g) C_g \tag{16}$$

Again, the separation of variables in well formed covering trees allows for grouping the sets of variables $CH(g)$:

$$\bigodot_{h\in\tilde{T}} h \odot g_1 \odot \left[\xi(DV(T)\setminus CH(g_1)) \bigodot_{g\in L(T)\setminus g_1} C_g\right] \odot \xi(\bigcup_{g\in succ(g_1)} CH(g)) \bigodot_{g\in succ(g_1)} C_g \tag{17}$$

The lemma follows now from a last grouping again allowed by the separation of variables property and from 13 and 14.□

The resolution tree is a full pruning of itself. So 9 applies. The remaining C_g in formula 9 correspond to leaves of R and the corresponding variables $CH(g)$, if not void, are variable leaves of the well covering tree not in $X|_R$. Since C_g are leaves of R, $C_g = g$ and all $C_g|_{\bar{X}|_R}$ have separated variables and are satisfied by $\bar{X}|_{CH(g)}$ in the resolution process. These last set of variable instantiations can be grouped together with $\bar{X}|_R$ to give the $\bar{X}$ instantiation of X obtained in the resolution process.□

This result shows that the initial constraint problem is transformed in a partially ordered set of simpler constraint problems. This transformation can be considered as

a *reparametrization* of the initial problem. More over, the partial order is a resolution order similar to the resolution order obtained when ones has put a system of linear equation into triangular form. However, the resolution order associated with a triangular form is linear. On the opposite the resolution order obtained from our algorithm is partial and given by a tree.

16 Conclusion

We have presented a family of decision algorithms well suited for loosely coupled constraint problems in a broad acceptance. This is based on a mixed traversal of a bipartite graph: the dependence graph. Moreover, the well formed covering trees give not only a decision procedure but also a true solver and a reparametrization of loosely coupled constraint problems into partially ordered simple problems.

Further developments are under study on these algorithms:

- in the building of well formed covering trees some degrees of freedom remains and may be used for optimization.
- when using the algorithm as a decision procedure, inconsistency may be detected earlier than at the end of traversal and used to find local inconsistencies.
- in finite domains, optimization constraints can be transformed into set constraints. We plan to use the reparametrization to transform loosely coupled optimization constraints into loosely coupled set constraints.

As already mentioned, the boolean constraint decision procedure and solver are implemented as a module of the BIOQUALI package. Implementations of optimization solvers are underway since there are needed for the development of BIOSIGNAL language and environment.

References

[1] Frank R. Kschischang, Brendan J. Frey, Hans-Andrea Loeliger. Factor Graphs and the Sum-Product Algorithm. *IEEE Transactions on Information theory, vol 47, no2, February 2001*

[2] Eric Fabre. Bayesian Networks of Dynamic Systems. *Document d'habilitation a diriger les recherches. Universite de Rennes. IRISA/INRIA 2007*

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Futurs : Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399